7 I/O-automata Based Testing

Machiel van der Bijl[1] and Fabien Peureux[2]

[1] Software Engineering
Department of Computer Science
University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
vdbijl@cs.utwente.nl
[2] Laboratoire d'Informatique (LIFC)
Université de Franche-Comté - CNRS - INRIA
16, route de Gray - 25030 Besançon, France
peureux@lifc.univ-fcomte.fr

7.1 Introduction

The testing theories on labeled transition systems, that we have seen so far, abstract from input and output actions. They only use the general concept "action" without a notion of direction. Although theoretically appealing, this may seem strange from a practical perspective. As a tester, we stimulate the system under test by providing inputs and observe its responses; the outputs of the system under test.

This chapter introduces the concepts from the conformance testing framework, as introduced in Part II (the section introducing "testing of labeled transition systems"), with the notion of inputs and outputs. We start with the introduction of three models for specifications and/or implementations in Section 7.3. After this we continue with several implementation relations in Section 7.4. Next we show how these models and implementation relations can be put to practice in Section 7.5. In this section we treat the derivation and execution of test cases on a system under test. We finish with our conclusions in Section 7.6.

In this chapter we made a deliberate choice to restrict the number of theories presented. We choose these theories, that –in our opinion– are relevant to get a good introduction into the field of testing with inputs and outputs. As a result, the chapter is centered around work from the following people (presented more or less in historical order).

- Lynch and Tuttle, who introduced the Input-Output Automata model and several implementation relations that use this model [LT87],
- Segala, who extended may and must testing with inputs and outputs [Seg92],
- Phalippou, who introduced the Input-Output State Machine and several implementation relations that use this machine [Pha94b],
- Tretmans, who introduced the Input-Output Transition System model and showed that his framework of **ioco** unifies several implementation relations [Tre96b]. Furthermore, Tretmans is one of the few that actually used his input-output testing theory in practice and therefore we use the **ioco** theory as the main example for test derivation and test execution,

M. Broy et al. (Eds.): Model-Based Testing of Reactive Systems, LNCS 3472, pp. 173-200, 2005.

- Petrenko, who developed a theory to test Input-Output Automata in a similar way as Finite State Machines [TP98].

Although we do not treat symbolic testing in this chapter, we want to mention that Rusu et al. developed a theory to enable symbolic test generation for Input-Output Automata [RdBJ00].

7.2 Formal Preliminaries

In this section we introduce some standard notation for labeled transition systems. People that are familiar with this notation can skip this section.

Labeled Transition Systems. A labeled transition system (LTS) description is defined in terms of states and labeled transitions between states, where the labels indicate what happens during the transition. Labels are taken from a global set $\mathbf{L}$. We use a special label $\tau \notin \mathbf{L}$ to denote an internal action. For arbitrary $L \subseteq \mathbf{L}$, we use L_τ as a shorthand for $L \cup \{\tau\}$. We deviate from the standard definition of labeled transition systems in that we assume the label set of an LTS to be partitioned in an input and an output set and that the LTS is rooted; see for example definition 22.1.

Definition 7.1. A *labeled transition system* is a 5-tuple $\langle Q, I, U, T, q_0 \rangle$ where Q is a non-empty countable set of *states*; $I \subseteq \mathbf{L}$ is the countable set of *input labels*; $U \subseteq \mathbf{L}$ is the countable set of *output labels*, which is disjoint from I; $T \subseteq Q \times (I \cup U \cup \{\tau\}) \times Q$ is a set of triples, the *transition relation*; $q_0 \in Q$ is the *initial state*.

We use L as shorthand for the entire label set ($L = I \cup U$); furthermore, we use Q_p, I_p etc. to denote the components of an LTS p. We commonly write $q \xrightarrow{\lambda} q'$ for $(q, \lambda, q') \in T$. Since the distinction between inputs and outputs is important, we sometimes use a question mark before a label to denote input and an exclamation mark to denote output. We denote the class of all labeled transition systems over I and U by $\mathcal{LTS}(I, U)$. We represent a labeled transition system in the standard way, by a directed, edge-labeled graph where nodes represent states and edges represent transitions.

A state that cannot do an internal action is called *stable*. A state that cannot do an output or internal action is called *quiescent*. We use the symbol δ ($\notin \mathbf{L}_\tau$) to represent quiescence: that is, $p \xrightarrow{\delta} p$ stands for the absence of any transition $p \xrightarrow{\lambda} p'$ with $\lambda \in U_\tau$. For an arbitrary $L \subseteq \mathbf{L}$, we use L_δ as a shorthand for $L \cup \{\delta\}$.

An LTS is called *strongly responsive* or *strongly convergent* if it always eventually enters a quiescent state; in other words, if it does not have any infinite U_τ-labeled paths. For technical reasons we restrict $\mathcal{LTS}(I, U)$ to strongly responsive transition systems.

A *trace* is a finite sequence of observable actions. The set of all traces over L ($\subseteq \mathbf{L}$) is denoted by L^*, ranged over by σ, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2. We use

the standard notation with single and double arrows for traces: $q \xrightarrow{a_1 \cdots a_n} q$ denotes $q \xrightarrow{a_1} \cdots \xrightarrow{a_n} q'$, $q \stackrel{\epsilon}{\Longrightarrow} q'$ denotes $q \xrightarrow{\tau \cdots \tau} q'$ and $q \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} q$ denotes $q \stackrel{\epsilon}{\Longrightarrow} \xrightarrow{a_1} \stackrel{\epsilon}{\Longrightarrow} \cdots \xrightarrow{a_n} \stackrel{\epsilon}{\Longrightarrow} q'$ (where $a_i \in \mathbf{L}_{\tau\delta}$).

We will not always distinguish between a labeled transition system and its initial state. We will identify the process $p = \langle Q, I, U, T, q_0 \rangle$ with its initial state q_0, and we write, for example, $p \stackrel{\sigma}{\Longrightarrow} q_1$ instead of $q_0 \stackrel{\sigma}{\Longrightarrow} q_1$.

Below we give some often used definitions for transition systems. $init(p)$ is the set of actions for which there is a transition in p, p **after** σ is the set of states that can be reached by performing the trace σ in p, $out(p)$ is the set of output actions for which there is a transition in p. Finally, the set of *suspension traces* of an LTS p, or $Straces(p)$ for short, is the set of traces over the label set L_δ that are possible in p.

Definition 7.2. Let $p \in \mathcal{LTS}(I, U)$, let $P \subseteq Q_p$ be a set of states in p and let $\sigma \in \mathbf{L}_\delta^*$.

(1) $init(p) =_{\text{def}} \{\mu \in L_\tau \mid p \xrightarrow{\mu}\}$
(2) p **after** $\sigma =_{\text{def}} \{\, p' \mid p \stackrel{\sigma}{\Longrightarrow} p' \,\}$
(3) P **after** $\sigma =_{\text{def}} \bigcup \{\, p$ **after** $\sigma \mid p \in P \,\}$
(4) $out(p) =_{\text{def}} \{x \in U_\delta \mid p \xrightarrow{x}\}$
(5) $out(P) =_{\text{def}} \bigcup \{\, out(p) \mid p \in P \,\}$
(6) $Straces(p) =_{\text{def}} \{\sigma \in L_\delta^* \mid p \stackrel{\sigma}{\Longrightarrow}\}$

7.3 Input Output Automata

We start with the introduction of several models for the specification and/or implementation as explained in the conformance testing framework, introduced in Part II. In model-based testing with inputs and outputs, input-output automata are a popular model. Several models with inputs and outputs have been proposed and all of these are quite similar. In this section we introduce the following three models. We will use these models in the rest of this chapter.

- Input Output Automata (IOA) as introduced by Lynch and Tuttle [LT89].
- Input Output State Machines (IOSM) as introduced by Phalippou [Pha94b].
- Input Output Transition Systems (IOTS) as introduced by Tretmans [Tre96c].

The general notion underlying all of these models is the distinction between actions that are locally controlled and actions that are not locally controlled. The output and internal actions of an automaton are locally controlled. This means that these actions are performed autonomously, i.e., independent of the environment. Inputs on the other hand, are not locally controlled; they are under control of the environment. This means that the automaton can never block an input action; this property is called input-enabledness or input completeness.

Input-output automaton. The input-output automaton is the first model with the notion of input completeness. It was introduced by Lynch and Tuttle in 1987 [LT87]. After this paper, they wrote a paper dedicated to input-output

automata [LT89]. An automaton's actions are classified as either 'input', 'output' or 'internal'. Communication of an IOA with its environment is performed by synchronization of output actions of the environment with input actions of the IOA and vice versa. Because locally controlled actions are performed autonomously, it requires that input actions can never be blocked. Therefore an IOA is input enabled (it can process all inputs in every state).

Definition 7.3 (I/O automaton)**.**
An input-output automaton $p = \langle sig(p), states(p), start(p), steps(p), part(p)\rangle$ is a five-tuple, where

- $sig(p)$ is the action signature. Formally an action signature $sig(p)$ is a partition of a set $acts(p)$ of actions into three disjoint sets: $in(p)$ input actions, $out(p)$ output actions and $int(p)$ internal actions.
- $states(p)$ is a countable, non-empty set of states.
- $start(p) \subseteq states(p)$ is a non-empty set of start states.
- $steps(p) \subseteq states(p) \times acts(p) \times states(p)$ is the transition relation with the property: $\forall a \in in(A), q \in states(p) : q \xrightarrow{a}$. This means that for every state q, there exists a state q', such that for every input action a, there is a transition $(q, a, q') \in steps(p)$ (input enabledness).
- $part(p)$ is an equivalence relation that partitions the set $local(p) = int(p) \cup out(p)$ of locally controlled actions into at most a countable number of equivalence classes.

The signature partitions the set of actions into input, output and internal actions. The input actions are actions *from* the environment, the output actions are actions *to* the environment and internal actions are actions that are *not observable* by the environment. The transition relation relates the actions to the states; by performing an action the automaton goes from one state to another. A possible problem with the input-output automata model is that an automaton cannot give an output action, because it has to handle a never ending stream of input actions. Since it is input-enabled it will synchronize on an input from the environment. Lynch and Tuttle therefore introduce the notion of fairness for IOA. In short this means that a locally controlled action cannot be blocked by input actions forever. This is the reason that the set $local(p)$ is introduced. The partitioning $part(p)$ of the locally controlled actions is used in the operationalization of fair testing. We will treat fairness in more detail in Section 7.4.1. Note that the problem of fair testing exists for all models that implement the notion of *input enabledness*. IOA implement strong input enabledness. This is formally defined by $\forall a \in in(p), q \in states(p) : q \xrightarrow{a}$. For weak input enabling it is also allowed to perform a number of internal actions before the input action can be performed: $\forall a \in in(p), q \in states(p) : q \stackrel{a}{\Longrightarrow}$.

Example. In Figure 7.1 we show three transition systems: an IOA (left), an IOSM (middle) and an IOTS (right). We will discuss the IOSM and the IOTS later on, now we focus on the IOA. The IOA represents a coffee machine. We can push two buttons: button1 and button2. After pushing button1 the machine

initializes and outputs coffee, and after pushing button2 the machine initializes and outputs tea. button1 and button2 are input actions, coffee and tea are output actions and init is an internal action. To make the picture easier to read, we have abbreviated button1 and button2 to $b1$ and $b2$ respectively. The self-loops with $b1$ and $b2$ in states q_1 till q_6 show that the automaton is input enabled in every state. q_0 does not need these self loops, since button1 and button2 are already enabled in this state.

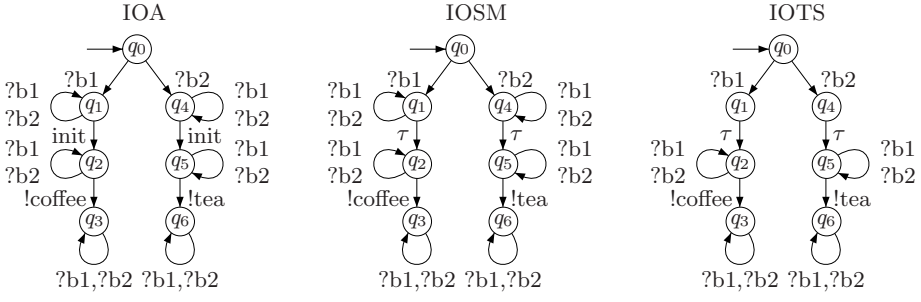


Fig. 7.1. Examples of an IOA, IOSM and IOTS

Input output state machine. This is the model as introduced by Phalippou [Pha94b]. The model is quite similar with the input-output automaton.

Definition 7.4 (input-output state machine (IOSM))**.**
An input-output state machine is a 4-tuple $M = \langle S, L, T, s_0 \rangle$ where:

- S is a finite, non-empty set of states.
- L is a finite, non-empty set of labels.
- $T \subseteq S \times ((\{?, !\} \times L) \cup \{\tau\}) \times S$ is the transition relation. Every element of T is a transition between a source and a target state. The associated action is either observable (input (?) or output (!)) or internal, denoted by τ. Furthermore, every state is strongly input-enabled.
- $s_0 \in S$ is the start state of the state machine.

These automata model systems whose operation can be interpreted in the following way:

- transition $(s_1, !a, s_2)$: the automaton, which is in the state s_1, performs the interaction a and goes to the state s_2. The decision to start the transition is local to the automaton.
- transition $(s_1, ?a, s_2)$: the automaton, which is in the state s_1, receives the interaction a and goes to the state s_2. The decision to start the transition is external to the automaton, since the transition is started when the interaction is received.
- transition (s_1, τ, s_2): the automaton, which is in the state s_1, goes to the state s_2 after an internal decision, without performing any observable interaction.

The main difference between an IOSM and an IOA is that there is no equivalence relation in the IOSM definition. Furthermore internal actions are abstracted into one action τ. Another difference is that the sets of states, input labels and output labels are restricted to be finite. Phalippou also uses the notion of locally (output and internal) and exteriorly (input) controlled actions and strong input-enabledness.

Example. In Figure 7.1, the transition system in the middle is an IOSM. It is very similar to the IOA on the left. The only difference is that the internal action 'init' is replaced by τ.

Input output transition system. This is the model as introduced by Tretmans [Tre96c].

Definition 7.5. An *input-output transition system* is a 5-tuple $\langle Q, I, U, T, q_0 \rangle$ where

- Q is a countable, non-empty set of *states*.
- I is a countable set of *input labels*.
- U is a countable set of *output labels*, such that $I \cap U = \emptyset$.
- $T \subseteq Q \times (I \cup U \cup \{\tau\}) \times Q$ is the *transition relation*, where $\tau \notin I, \tau \notin U$. Furthermore, every state is weakly input-enabled: $\forall\, q \in Q, a \in I : q \stackrel{a}{\Longrightarrow}$.
- $q_0 \in Q$ is the *start state*.

The input-output transition system (IOTS) is a more general version of the IOSM. Like the IOSM it does not have the equivalence relation of the IOA and it also models internal actions with the τ label. However, it does not restrict the set of states and labels to be finite. Furthermore, there is a clean partitioning of the label set in inputs and outputs, where the IOSM hides this in the transition relation. A subtle but important difference with IOA is that an IOTS is weakly input enabled: $\forall a \in I, q \in Q : q \stackrel{a}{\Longrightarrow}$. We denote the class of input-output transition systems over I and U by $\mathcal{IOTS}(I, U)$.

Example. In Figure 7.1, the transition system on the right is an IOTS. We see that the internal action init is replaced by τ. Notice furthermore that the states q_1 and q_4 do not have the self-loops with button1 and button2. This is allowed because an IOTS is weakly enabled. With an internal action we can go from q_1 to the input enabled state q_2 (note that the same holds for q_4 and q_5).

7.4 Implementation Relations with Inputs and Outputs

In this section, we will introduce a number of implementation relations. As was introduced in the conformance testing framework in Section II, an implementation relation (or conformance relation) is a relation that defines a notion of correctness between an implementation and a specification. When the implementation relation holds we say that the specification is implemented by the implementation or, in other words, that the implementation conforms to the

specification. Several implementation relations have been defined for the automata that were introduced in the previous section. In this section, we start with implementation relations defined on IOA and continue with implementation relations on labeled transition systems.

7.4.1 Preorders on IOA

In this section, we treat implementation relations on Input Output Automata. Some of these implementation relations can also be expressed on labeled transition systems as we will explain in the next section. All of the implementation relations that we treat in this section are preorders.

We first recapitulate some concepts that are used with IOA. An *execution fragment* of an IOA p is an alternating, possibly infinite sequence of states and actions $\alpha = q_0 a_1 q_1 a_2 q_2 \cdots$ such that $(q_i, a_{i+1}, q_{i+1}) \in steps(p)$. When q_0 is a start state of p we call the execution fragment an execution of p. An external trace of an IOA p is an execution (fragment) that is restricted to the set of external actions. We use the notation $etraces(p)$ to denote the external traces of IOA p, where $etraces^*(p)$ denotes the set of finite external traces of p. We use the notation $a \in enabled(q)$ to denote that state q enables a transition with action a. This means that there is a state q' for which $(q, a, q') \in steps(p)$. To denote the set of enabled external actions in a state q, we use the notation $wenabled(q)$. An IOA p is finitely branching iff each state of p enables finitely many transitions.

We start with the trace inclusion preorder. This is a very weak relation. It expresses that one system is an implementation of the other if its set of external traces is a subset of the set of external traces of the specification.

Definition 7.6 (External trace inclusion)**.** For IOA i and s:

$$i \leq_{tr} s =_{\text{def}} etraces(i) \subseteq etraces(s)$$

The above definition is defined in a so-called intentional way. Many implementation relations can also be defined in an extensional way in the style of De Nicola and Hennessy [NH84]. The term *extensional* refers to an external observer. The intuition behind this idea is that an implementation conforms to a specification if no external observer can see the difference. We will not use the extensional definition in this section, but we refer to Chapter 5 for more information about extensional definitions of implementation relations and to [Tre96b] for more information on extensional definitions of implementation relations with inputs and outputs.

Example. We give an example of the external trace inclusion preorder in Figure 7.2. On the left hand side we see a specification of a coffee machine. It prescribes that after pressing the button at least twice we expect to observe either coffee or tea as output. We will reuse this coffee machine specification in other examples. On the right hand side we see two implementations. The first

implementation does not implement coffee as an output. It is still correct, because the set of traces of the implementation is a subset of the set of traces of the specification, even with the trace $button{\cdot}button{\cdot}button^*{\cdot}coffee{\cdot}button^*$ missing. External trace inclusion is not a very realistic implementation relation, because it also approves implementations that are intuitively incorrect. For example, the implementation on the right only implements the pushing of the button, without serving any drink. This is correct, because the set of traces $button{\cdot}button{\cdot}button^*$ is a subset of the external traces of the specification.

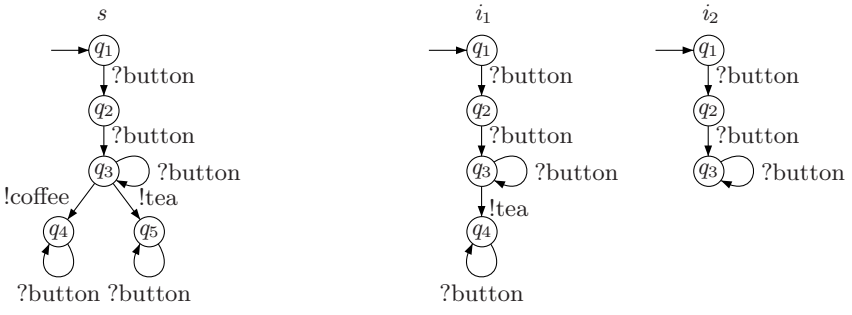


Fig. 7.2. Example of the external trace inclusion preorder

Lynch and Tuttle introduced the notion of fair execution for IOA. Remember that IOA are (strong) input enabled. This means that an infinite stream of input actions can prevent an output or internal action from occurring. Intuitively the idea behind fair execution is that locally controlled actions cannot be blocked by input actions for ever. This is expressed formally in the definition below.

The definition uses the concept of *quiescent* executions. Similar to transition systems, for IOA an execution is quiescent if it ends in a quiescent state, i.e., a state that can only perform input actions (so no locally controlled actions). A quiescent trace, is a trace that leads to a quiescent state. The set of quiescent traces is the set of finite external traces that lead to a quiescent state : $qtraces(p) = \{\sigma \in etraces^* p^* \mid \exists q \in states(p) : p \stackrel{\sigma}{\Longrightarrow} q \wedge enabled(q) = in(p)\}$.

An execution α of an IOA p is *fair* if either α is *quiescent* or α is infinite and for each class $c \in part(p)$ either actions from c occur infinitely often in α or states from which no action from c is enabled appear infinitely often in α. A fair trace of an IOA p is the external trace of a fair execution of p. The set of fair traces of an IOA p is denoted by $ftraces(p)$. Given the notion of fair traces we can define a preorder over the sets of fair traces of IOA.

Definition 7.7 (Fair preorder)**.** Given two IOA's i and s with the same external action signature, the fair preorder is defined as:

$$i \sqsubseteq_F s \Leftrightarrow ftraces(i) \subseteq ftraces(s).$$

We will give examples of the fair preorder a little later in this section, because we first want to introduce a preorder that is strongly related to the fair preorder,

namely the quiescent preorder introduced by Vaandrager [Vaa91]. It uses the concept of quiescent traces, introduced above.

Definition 7.8 (Quiescent preorder)**.** Given two IOA's i and s with the same external action signature, the quiescent preorder is defined as:

$$i \sqsubseteq_Q s \Leftrightarrow etraces^*(i) \subseteq etraces^*(s) \wedge qtraces(i) \subseteq qtraces(s).$$

The fair and quiescent preorders look much alike, but there are some important differences. The quiescent preorder uses finite traces to test for trace inclusion, whereas the fair preorder includes infinite traces. The relation between the two preorders is easiest explained with an example (the examples are reused with kind permission of Segala [Seg97]).

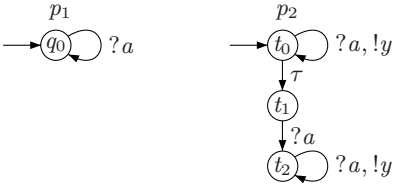


Fig. 7.3. Quiescent versus fair preorder, example 1

Example. Figure 7.3 shows two IOA p_1 and p_2, a is an input action, y is an output action and τ is an internal action. The partition of locally controlled actions for both IOA is a single class $\{y, \tau\}$. Let us first illustrate the set of external and quiescent traces. For p_1 we have $etraces(p_1) = qtraces(p_1) = a^*$, meaning a set of zero or more occurrences of a. For p_2 we have $etraces(p_2) = \{a, y\}^* \cdot a \cdot \{a, y\}^*, qtraces(p_2) = \{a, y\}^*$. With $\{a, y\}^*$ we mean an arbitrary number of times, an arbitrary number of a's followed by an arbitrary number of y's (or vice versa), like *aayayya*. Regarding the fair traces, for p_1 it is trivial that each finite sequence a^n is quiescent and therefore a fair trace. Also for p_2, the finite sequence a^n is a quiescent and fair trace. After looping n times in t_0 we move to t_1 by a τ transition. Therefore $p_1 \sqsubseteq_Q p_2$. However, the sequence a^ω (infinite times a) is a fair trace of p_1 but not of p_2. The latter is because, we are either infinitely often in t_0 or t_2 but neither $\{\tau, y\}$ is not enabled, nor $\{\tau, y\}$ is occurring infinitely often. Thus $p_1 \not\sqsubseteq_F p_2$.

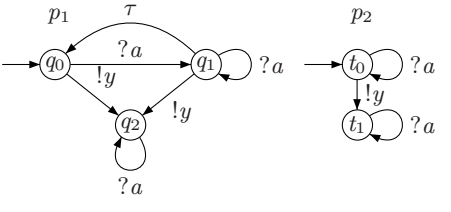


Fig. 7.4. Quiescent versus fair preorder, example 2

Example. Another similar difference is illustrated in Figure 7.4. We have the IOA p_1 and p_2, both can perform an arbitrary number of a input actions followed by one y output action, followed again by an arbitrary amount of a actions. p_1 and p_2 are equivalent according to the quiescent preorder (both $p_1 \sqsubseteq_Q p_2$ and $p_2 \sqsubseteq_Q p_1$), because they have the same external traces and their quiescent traces contain at least a y action. However, p_1 and p_2 are not equivalent according to the fair preorder, when considering the same partitioning as before: $\{y, \tau\}$. This is because a^ω is a fair trace of p_1 but not of p_2. This might not be easy to see at first glance, but remember that the partition of locally controlled actions is $\{y, \tau\}$. In p_1 we can do the fair execution $(q_0 \cdot a \cdot q_1 \cdot \tau q_0)^\omega$ (and thus the fair trace a^ω).

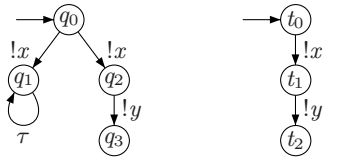


Fig. 7.5. Quiescent versus fair preorder example 3

Divergence (i.e., the possibility for a system of to do an infinite number of internal transitions) shows another difference between the quiescent and fair preorder. Because of divergence, a fair trace is not necessarily a quiescent trace, as is illustrated in the next example.

Example. In Figure 7.5, we have two IOA p_1 and p_2 with output actions x and y. According to the quiescent preorder both automata are equivalent. But they are not equivalent according to the fair preorder, since x is a fair trace of p_1, but not of p_2.

As we can see from these examples, the fair preorder is a stronger relation than the quiescent preorder. Because of the definition of the quiescent preorder, this is not very surprising. As a side step, Segala shows a way to make the quiescent preorder and the fair preorder equivalent by making some restrictions on the IOA's that we allow [Seg97]. Basically, the property we are looking for is that we can approximate an infinite fair trace by a finite trace and can extend a finite fair trace to an infinite fair trace. This is expressed by the properties *fair continuity* and *fair approximability*. An IOA p is fair continuous if the limit of any chain of fair traces of p is also a fair trace. Fair continuity is nicely illustrated in Figure 7.3. The sequence a^n is a fair trace of p_2, but if we take n to infinity this is not the case. In other words the trace a^n is not fair continuous. An IOA p is fair approximable if each infinite trace of p is the limit of a chain of fair traces of p. This is illustrated in Figure 7.4. The trace a^ω is a fair trace of p_1 but it is not fair approximable because the finite trace a^n is not fair.

Under the restrictions of fair approximability and fair continuity we can show that the fair preorder and the quiescent preorder are equivalent for strongly converging IOA (we need strong convergence to rule out divergence).

Theorem 7.9. *Let $i,\ s \in IOA$ be strongly convergent.*

$$i \sqsubseteq_F s \Rightarrow i \sqsubseteq_Q s$$

If p_1 is fair approximable, and p_2 is fair continuous, then $p_1 \sqsubseteq_Q p_2 \Rightarrow p_1 \sqsubseteq_F p_2$

In the next part of this section we will introduce may and must testing for systems with inputs and outputs. First we quickly recapitulate some of this theory. The method for comparing transition systems that was initiated by De Nicola and Hennessy is based on the observation of the interactions between a transition system and an external experimenter as introduced in Chapter 5. An experimenter e for a transition system p is a transition system that is compatible with p. The input actions of e are the output actions of p ($in(e) = out(p)$) and the output actions of e are the input actions of p, plus an action w called the success action ($out(e) = in(p) \cup \{w\}$). The experimenter e runs in parallel with p and synchronizes its output actions with input actions of p and vice versa (except w). An experiment x is an execution of $p \| e$ which is infinite or ends in a deadlocked state. We say that the experiment is successful if w is enabled in at least one state of the execution x. If there is a successful experiment of $p \| e$ we use the notation p **may** e. If every experiment of $p \| e$ is successful we use the notation p **must** e. On this notion of may and must we can define preorder relations. We will start with the may preorder.

Definition 7.10 (MAY preorder)**.** Let $i,\ s \in IOA$:

$$s \sqsubseteq_{\text{MAY}} i \Leftrightarrow \forall e : s \textbf{ may } e \Rightarrow i \textbf{ may } e$$

Hennessy has shown that the may preorder and external trace inclusion are equivalent [Hen88].

Theorem 7.11. *Let $i,\ s \in IOA$: $s \sqsubseteq_{\text{MAY}} i \Leftrightarrow etraces(s) \subseteq etraces(i)$*

For the must preorder we need a little more work. Segala uses the following definition of the must relation [Seg97]:

Definition 7.12 (MUST)**.** Given an IOA p, a set of states Q_1 and a set of external actions A.
Q_1 **must** $A \Leftrightarrow$

(1) $A \cap in(p) \neq \emptyset$, or
(2) for each $q \in Q_1$:
    (a) wenabled$(q) \cap out(p) \subseteq A$, and
    (b) wenabled$(q) \cap A \neq \emptyset$

With this definition of the must relation on IOA we define the must preorder in the following way.

Definition 7.13 (MUST preorder)**.** Let $i,\ s \in IOA$:

$$s \sqsubseteq_{\text{MUST}} i \Leftrightarrow \forall \sigma \in ext(s)^*, A \subseteq ext(s) : s \textbf{ after } \sigma \textbf{ must } A \Rightarrow i \textbf{ after } \sigma \textbf{ must } A$$

Segala has shown that this definition of the must preorder is equivalent with the quiescent preorder. The only restriction we need is that the IOA are strongly converging and finitely branching.

Theorem 7.14. *Let i and s be finitely branching and strongly convergent IOA.*

$$s \sqsubseteq_{\text{MUST}} i \Leftrightarrow i \sqsubseteq_Q s.$$

7.4.2 IOCO Based Testing

In this section, we introduce input-output variants of several pre-order based testing relations that were introduced in Chapter 5. All the input-output testing relations that we show in this section take an LTS with inputs and outputs as a specification and assume the implementation to be an IOTS. We show that all the implementation relations that we present in this chapter can be unified in the **ioco** implementation relation, hence the name of this chapter. In order to relate the implementation relations we use definitions that deviate from the original definitions. The equivalence of our definitions with the original definitions is proved in [Tre96b]. The way that our definitions differ is that we use an intentional characterization of the implementation relations, i.e., a characterization in terms of properties of the labeled transition systems themselves. This in contrast to an extensional characterization where an implementation relation is defined in terms of observations that an external observer can make. In the intentional characterization observations are expressed in possible outputs of the labeled transitions system after performing a certain trace.

We will introduce the following input output implementation relations:

- The input-output variant of testing preorder, denoted by $\leq_{iot}$.
- The input-output variant of the **conf** relation, denoted by **ioconf**.
- The input-output variant of refusal preorder, denoted by $\leq_{ior}$.
- The **ioco** implementation relation.

Input-output testing relation The first implementation relation that we introduce is the input-output testing relation. This is testing pre-order with a notion of input and output actions. The set of traces with which we test are in L^*. This means that we can use any trace to test with, even if its behavior is not specified by the specification.

Definition 7.15 (Input output testing relation)**.** Let $i \in \mathcal{IOTS}(I, U), s \in \mathcal{LTS}(I, U)$

$$i \leq_{iot} s =_{\text{def}} \forall \sigma \in L^* : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma).$$

We can read this definition in the following way. An implementation i is $\leq_{iot}$-correct with respect to a specification s if for all traces with which we test, the set of outputs of the implementation after such a trace is a subset of the set of outputs of the specification after the same trace. In terms of observations this means that we should not be able to observe different or more behavior from the implementation than from the specification. A possible output is the absence of output or quiescence. We use the meta-label δ to denote quiescence. It is interesting to know that the $\leq_{iot}$ relation is equivalent with the quiescent preorder (and thus with the must preorder) [Tre96b].

Example. We will illustrate the input output testing relation with the example of the trace inclusion preorder in Figure 7.2. The implementation i_1 is $\leq_{iot}$-correct with respect to specification s. We see that it does not implement the coffee output after pressing the button twice, so how can it be correct? Let us take a look at the definition of $\leq_{iot}$. The specification prescribes that the set of outputs after the trace $button{\cdot}button = \{coffee, tea\}$. When we take a look at i_1 we see that the set of outputs after $button{\cdot}button = \{tea\}$. Because $\{tea\} \subseteq \{coffee, tea\}$ it is correct behavior. So the intuition behind non deterministic output is that we do not care which branch is implemented as long as at least one is. We can do the same analysis for implementation i_2. Here we find that after pressing the button twice i_2 does not give any output; it is quiescent. This means that $out(i_2 \textbf{ after } button{\cdot}button) = \{\delta\}$. This is not a subset of $\{coffee, tea\}$ and there fore $i_2 \not\leq_{iot} s$. We see that $\leq_{iot}$ is a stronger implementation relation than external trace inclusion and furthermore one that agrees more with our intuition.

ioconf relation The input-output variant of the **conf** relation is called **ioconf** [Tre96b]. The difference with the input-output testing relation is that it uses a different set of traces to test with, namely the set of all possible traces of the specification: $traces(s)$. This means that we will not test behavior that is not specified. One way to interpret this is as *implementation freedom*: "We do not know or care what the implementation does after an unspecified trace". The advantage of this is that we can test with incomplete specifications. Since $traces(s) \subseteq L^*$, the **ioconf** relation is weaker than the $\leq_{iot}$ relation.

Definition 7.16 (ioconf). Let $i \in \mathcal{IOTS}(I, U), s \in \mathcal{LTS}(I, U)$

$$i \textbf{ ioconf } s =_{\text{def}} \forall \sigma \in traces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma).$$

Example. In Figure 7.6 we illustrate the **ioconf** relation. i_1 is the same implementation as in the examples for external trace inclusion and $\leq_{iot}$. This implementation is still correct under **ioconf**. This is easy to see, because the trace $button{\cdot}button \in traces(s)$; it is a trace of the specification and $out(i_1 \textbf{after} button{\cdot}button) = \{tea\} \subseteq out(s \textbf{after} button{\cdot}button) = \{coffee, tea\}$. Implementation i_2 introduces new behavior. When we kick the coffee machine it outputs soup. This kind of behavior is nowhere to be found in the specification; the behavior of kicking the machine is underspecified. This kind of behavior would be a problem for $\leq_{iot}$ since it will test on all possible behavior of the label

set: L^*. When we test the kicking of the machine with $\leq_{iot}$ we get the following result: $out(i_2 \textbf{ after } kick) = \{soup\} \not\subseteq out(s \textbf{ after } kick) = \emptyset$. This is the reason that with $\leq_{iot}$ we need a completely specified specification. Else we know up front that no implementation will conform to the specification. **ioconf** does not have this restriction, because it will only test behavior that is specified. Since $kick \notin traces(s)$, we will not test its behavior. Because all the other behavior of i_2 is identical to i_1 we have i_2 **ioconf** s. In case one does not like this kind of implementation freedom, the specification can be made complete and as a result the same testing power as the $\leq_{iot}$ relation is obtained.

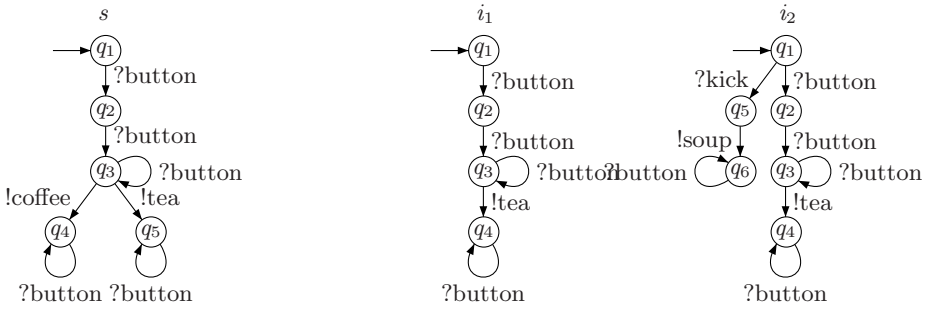


Fig. 7.6. Example of **ioconf**

Input-output refusal relation The next implementation relation, the input-output refusal relation, is the input output version of the refusal preorder (Chapter 5). What we saw with the $\leq_{iot}$ and **ioconf** implementation relations was that they both used traces that did not have δ's in them (no intermediate quiescence). It was only possible to observe quiescence at the end of a trace. The input-output refusal relation can do just that, it uses δ as an expected output in its set of traces to test with; so called repetitive quiescence. Quiescence can be seen as *refusal* to do an output action, hence the name of the implementation relation. We can see this as follows in the definition of the input-output refusal relation $\leq_{ior}$. The set of traces over which we test is: L^*_δ. Or, in other words, all possible combinations of actions from the label set with δ (quiescence). This means that it only makes sense to test with complete specifications as illustrated for $\leq_{iot}$ in Figure 7.6. Again the correctness criterion is that an implementation does not show more behavior than is allowed by the specification.

Definition 7.17 (Input output refusal)**.** Let $i \in \mathcal{IOTS}(I, U), s \in \mathcal{LTS}(I, U)$

$$i \leq_{ior} s =_{\text{def}} \forall \sigma \in L^*_\delta : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma).$$

We give an example of the $\leq_{ior}$ implementation relation together with **ioco**, because these two implementation relations are closely related.

ioco relation The **ioco** testing theory is named after its implementation relation **ioco**. The difference with the implementation relations that we have treated so

far lies again in the set of traces over which we test. Just like $\leq_{ior}$, **ioco** also uses the notion of quiescence. But the set of traces with which we test are the so-called *suspension traces* of the specification. These are the traces (with or without quiescence) that are specified in the specification. This set is smaller than the set of traces of $\leq_{ior}$. In other words, **ioco** is a weaker implementation relation than $\leq_{ior}$.

Definition 7.18 (ioco). Let $i \in \mathcal{IOTS}(I, U), s \in \mathcal{LTS}(I, U)$.

i **ioco** $s =_{\text{def}} \forall \sigma \in Straces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$.

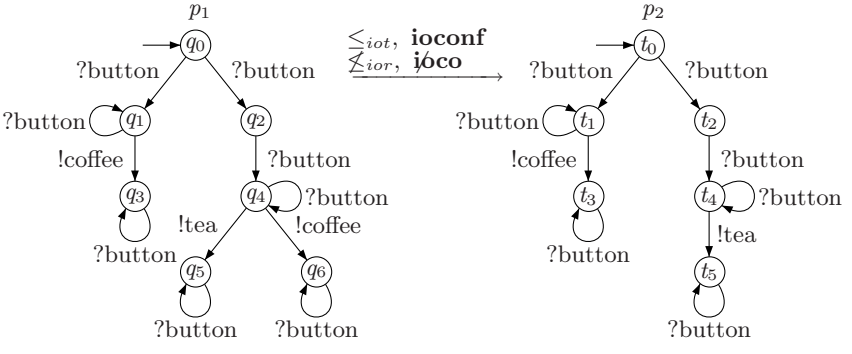


Fig. 7.7. Example of **ioco**

Example. We illustrate the $\leq_{ior}$ and **ioco** implementation relations in Figure 7.7. The example is reused with kind permission of Tretmans [Tre96b]. We see two IOTS's p_1 and p_2 that model a coffee machine with peculiar behavior. p_1 models a machine where after pressing a button once, you get either coffee or nothing (quiescence). If you got nothing and you press the button again you get either tea or coffee. p_2 models an almost identical machine, except after you press the button again after obtaining nothing after the first button press you will only get tea (so no coffee). If p_1 is the implementation and p_2 the specification we can see that $\leq_{iot}$ and **ioconf** hold, whereas $\leq_{ior}$ and **ioco** do not. Let us begin with **ioco**, **ioco** can see the difference between the transition systems because of the following trace. After $button{\cdot}\delta{\cdot}button$ we will observe tea and coffee for p_1, whereas p_2 prescribes that only tea is allowed. The same holds for $\leq_{ior}$ since it is also capable of this same test case. However $\leq_{iot}$ and **ioconf** are not capable of observing quiescence during testing and can therefore not tell the difference between the trace $button{\cdot}button{\cdot}coffee$ in the left branch of the transition system or in the right branch of the transition system. In other words, they are not powerful enough to see the difference.

When we take p_2 as the implementation and p_1 as the specification we see that all implementation relations identify the implementation as correct. This is logical since the only difference between p_1 and p_2 is that p_2 does not offer the

possibility of coffee in the right branch. This is correct, since the specification p_1 gives the choice between implementing either one (or both).

The difference between **ioco** and $\leq_{ior}$ is the same difference as between $\leq_{iot}$ and **ioconf**. **ioco** is capable of dealing with incomplete specifications, whereas $\leq_{ior}$ is not.

The input-output implementation relations that we have introduced so far can be easily related. The only variable is the set of traces over which we test. Based on the relations between the sets of tested traces we can relate the strength of the implementation relations. This is easy to see since $traces(s) \subseteq Straces(s), L^* \subseteq L^*_\delta$. For reasons of completeness we have also added the preorders of the previous section. We know that these are defined on IOA and not on LTS's and IOTS's, but these relations can be easily converted to each others realms. The fair testing preorder is not in this comparison. As far as we know, nobody has made a comparison between the fair testing preorder and the other implementation relations in this chapter. It is clear that the fair testing preorder is stronger than the quiescent preorder, but it is not clear to what extent the fair testing preorder is comparable to **ioco**.

Proposition 7.19. *Comparison of expressiveness of the implementation relations.*

$$\left\{ \begin{array}{l} \sqsubseteq_{\text{MAY}} \\ \leq_{tr} \end{array} \right\} \subset \leq_{ior} \subset \left\{ \begin{array}{l} \sqsubseteq_{Q} \\ \sqsupseteq_{\text{MUST}} \\ \leq_{iot} \\ \textbf{ioco} \end{array} \right\} \subset \textbf{ioconf}$$

7.4.3 Work Introduced by M. Phalippou

We present in this section the implementation relations used in the method introduced by M.Phalippou [Pha94b]. This method is defined on a particular model of automata: input/output state machine (see IOSM definition 7.4).

Moreover, we are only interested in the states which are reachable from the initial state by a finite number of transitions. We can thus remove the set of all the states which do not verify this condition, or we only use the connex graph of the automaton containing the initial state. In a more formal way, this connex component of an automaton is defined as follows.

Definition 7.20 (Connex component of IOSM)**.**
Let $\langle S, L, T, s_0 \rangle$ be an IOSM. The connex component of this IOSM containing the initial state is an IOSM $CC(\langle S, L, T, s_0 \rangle) = \langle S_C, L_C, T_C, s_{0C} \rangle$ defined by:

(1) $L_C = L$
(2) $s_{0C} = s_0$
(3) S_C is recursively defined by the rules:
  (a) $s_{0C} \in S_C$
  (b) if $s \in S_C$ and $(s, \mu, s') \in T$ then $s' \in S_C$
(4) $T_C = \{(s, \mu, s') \in T, s \in S_C\}$

The properties relating to testing only depend on the traces of the handled automata. In order to introduce a formal definition of the traces on IOSM, we firstly need to define IOSM sequence and opposite sequence.

Definition 7.21 (Sequence and opposite sequence)**.**
Given an IOSM $S = \langle S_s, L_s, T_s, s_0\rangle$ and a sequence $(\sigma = \mu_1 ... \mu_n) \in (\{!, ?\} \times L_s)^*$. The opposite sequence, noted $\overrightarrow{\sigma}$ is defined by the sequence generated from σ by reversing the output (!) and the input (?) in the different actions. The following properties about sequence and opposite sequence can now be introduced:

(1) (s_0, σ, s_n) iff $(\exists (s_i)_{1 \leq i < n} \in S_s^n)(\forall\, i, 1 \leq i \leq n)((s_{i-1}, \mu_i, s_i) \in T_s)$
(2) (S, σ, s_n) iff (s_0, σ, s_n)
(3) (s_0, ε, s_1) iff $s_0 = s_1$ or $(\exists\, n \geq 1)(s_0, \tau^n, s_1)$
(4) $(s_0, \overrightarrow{\mu}, s_1)$ iff $(\exists\, s_2, s_3 \in S_s)((s_0, \varepsilon, s_2) \wedge (s_2, \mu, s_3) \wedge (s_3, \varepsilon, s_1))$
(5) $(s_0, \overrightarrow{\sigma}, s_n)$ iff $(\exists (s_i)_{1 \leq i < n} \in S_s^n)(\forall\, i, 1 \leq i \leq n)((s_{i-1}, \overrightarrow{\mu}_i, s_i) \in T_s)$

Definition 7.22 (Trace)**.**
An observable trace of S is a sequence σ of observable actions such as $(\exists\, s_n \in S_s)(s_0, \overrightarrow{\sigma}, s_n)$. The set of all the traces of S is denoted by $Tr(S)$.

The concept of trace makes it possible to disregard internal action τ. Thus, a trace is an observable behavior, i.e. visible from the interface of the IOSM. It should be noted that the traces of an IOSM are the same ones as those of its connex component containing the initial state.

Property 7.23. $Tr(S) = Tr(CC(S))$

To illustrate implementation relations on IOSM, we will use the example of coffee machine. Its specification S is presented in figure 7.8.

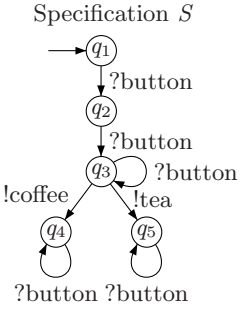


Fig. 7.8. IOSM specification of coffee machine

In the approach introduced by M. Phalippou, both the specification and the implementation to be tested are represented by an IOSM. Four possible implementations (I_1, I_2, I_3 and I_4) of the coffee machine are described in figure 7.9.

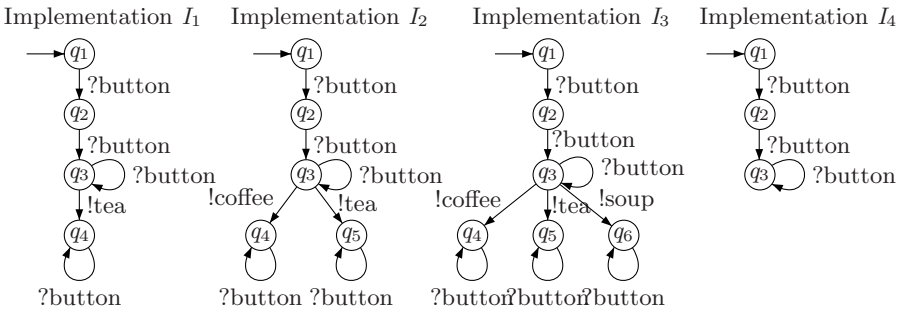


Fig. 7.9. Examples of IOSM implementation of coffee machine

The conformance is then defined as a relation, called implementation relation, between the implementation and its relevant specification. The next definition formally expresses this relation by means of IOSM as introduced by M. Phalippou.

Definition 7.24 (Implementation relation on IOSM)**.**
An implementation relation on IOSM is a relation R on $IOSM \times IOSM$. Given an implementation I and a specification S such as $I, S \in IOSM$, if $R(I, S)$ holds then we say that I conforms to S.

The choice of an implementation relation is generally arbitrary, although some minimal properties have to be respected according to the conformance objectives [PBD93]. To elaborate such relations, we place ourselves in a testing situation where all that we can do is to send interactions towards a black box system to be tested, and to analyze the outputs returned by the black box.

Definition 7.25 (Outputs authorized by the specification)**.**
Given $\sigma \in Tr(S)$ and L a finite non empty set of labels, $O = (\sigma, S) = \{a \in L \mid \sigma!a \in Tr(S)\}$ denotes the set of all the outputs authorized by the specification S after the trace σ.

All the definitions needed to present implementation relations on IOSM are now described. M. Phalippou defines five implementation relations adapted to the IOSM (these examples illustrate the variety of the arbitrary choices) [Pha93].

A first idea, to ensure that an implementation conforms to a specification, consists in verifying that the outputs returned by the implementation never contradict what is envisaged by the specification when something is envisaged. The goal of applying this kind of implementation relation, is not to know what it occurs when interactions, that are not specified by the specification, are send to the implementation. This implementation relation is known as R_1.

Definition 7.26 (Relation R_1)**.**
$R_1(I, S)$ iff $(\forall\, \sigma \in Tr(S))(\sigma \in Tr(I) \;\Rightarrow\; O(\sigma, I) \subseteq O(\sigma, S))$

According to the implementation relation R_1 and among the implementations of the figure 7.9, only I_3 is considered not to be in conformance with the specification of the figure 7.8. Indeed, the relation R_1 authorizes an implementation to return no output even if the specification envisages one or more possible behavior (see for example I_4). To avoid this lack, M. Phalippou thus consider a new implementation relation called R_2.

Definition 7.27 (Relation R_2)**.**
$R_2(I, S)$ iff
$(\forall \sigma \in Tr(S))(\sigma \in Tr(I) \Rightarrow \{O(\sigma, I) \subseteq O(\sigma, S) \wedge (O(\sigma, I) = \emptyset) \Leftrightarrow (O(\sigma, S) = \emptyset)\})$

This new implementation relation does not change the conformance of the implementation I_1 and I_2, and the non-conformance of I_3, but the implementation I_4 does not conform any more to the specification S. Indeed, according to R_2, an implementation conforms to a specification if the implementation gives less possible outputs than the specification does. But, this view could not be strong enough: it could be expected that the implementation must at least have all the capacities envisaged by the specification (but has freedom to make some more). The two following relations R_3 and R_4 express this idea : according to R_3 and R_4, I_1 does not conform to S while I_3 does.

Definition 7.28 (Relation R_3)**.**
$R_3(I, S)$ iff $Tr(S) \subseteq Tr(I)$

Definition 7.29 (Relation R_4)**.**
$R_4(I, S)$ iff $Tr(S) \subseteq Tr(I) \wedge (\forall \sigma \in Tr(S))((O(\sigma, I) = \emptyset) \Leftrightarrow (O(\sigma, S) = \emptyset))$

Finally, we can choose to require that the implementation makes exactly what is envisaged by the specification. The relation R_5 is built on this principle: it is built in fact by the conjunction of the implementation relations R_1 and R_3 (or R_2 and R_4). Using this last implementation relation, only I_2 conforms to the specification S.

Definition 7.30 (Relation R_5)**.**
$R_5(I, S)$ iff $(\forall \sigma \in Tr(S))(\sigma \in Tr(I) \wedge (O(\sigma, S) = O(\sigma, I)))$

It should be noted that the implementation relations R_1, R_2, R_3 and R_4 are expressed as preorder relations (see section 7.4.1). But, the most studied relation about Input Output Automata is the equivalence relation. When input complete specifications are used, the equivalence relation has to be modified, and we naturally obtain the relation R_5 introduced by M. Phalippou. Therefore, this last implementation relation appears to be a major result in the domain of Input Output Automata based testing. For example, G. Luo, A. Petrenko and G. Bochmann used an implementation relation similar to R_5 in order to select test cases from nondeterministic Finite State Machine [LvBP94].

7.5 Testing Transition Systems

In the previous section, we have discussed several implementation relations. In this section, we introduce two more concepts of the conformance testing framework, namely test derivation and test execution. We will show the relation between the conformance relation and test generation and execution. The **ioco** conformance relation is one of the few relations that is used for testing in practice. Apparently the other relations are more used for verification than testing. Therefore we use the **ioco** implementation relation as the example implementation relation for the test derivation and test execution sections. For more information about the practical application of the **ioco** theory we refer to Chapter 14.

Test cases Before we introduce test derivation, we first explain what a test case is; see also Section 20. A test case is a specification of the experiment that an experimenter wants to conduct on an implementation. A test case can be described by an LTS. In order to test according to implementation relations that have the notion of quiescence we introduce a new label in the label set of the tester: θ; θ is the tester's counter part of δ. With the θ label the test case can observe quiescence. So test cases will be in the domain $\mathcal{LTS}(U \cup \theta, I)$. We will add a couple of restrictions to the behavior of a test case. To guarantee that a test case finishes in finite time it should have finite behavior. Furthermore to ensure maximal control over the testing process we do not allow non-deterministic behavior. We also do not allow choice between multiple input actions and between input actions and output actions. This implies that a state in a test case is either a terminal state, or a state that offers exactly one input to the implementation or accepts all outputs of the implementation. To give a verdict over the success of the test case we label terminal states with **pass** and **fail**. These restrictions are formally expressed in the following definition of a test case. Note that a test case could in principle be defined without these restrictions. It could be an arbitrary LTS that synchronizes on the actions of the implementation under test. The definition we introduce here has shown to be both theoretically and practically useful.

Definition 7.31 (Test case)**.**

- An LTS $t = \langle Q, U \cup \{\theta\}, I, T, q_0 \rangle \in \mathcal{LTS}(U \cup \{\theta\}, I)$ is a test case if:
  - t is deterministic and has finite behavior. t is deterministic if $\forall \sigma \in L^*_\theta, p$ **after** σ has at most one element.
  - Q contains terminal states **pass** and **fail**, with $init(\textbf{pass}) = init(\textbf{fail}) = \emptyset$.
  - For any state $q \in Q$ of the test case, if $q \neq \textbf{pass}, \textbf{fail}$ then either $init(q) = \{a\}$ for some $a \in I$, or $init(q) = U \cup \{\theta\}$.
- The class of test cases over U and I is denoted as $\mathcal{TEST}(U, I)$.
- A test suite T is a set of test cases: $T \subseteq \mathcal{TEST}(U, I)$.

In the definition of a test case we can see that the label sets of the specification are reversed: Inputs of the specification are outputs of the implementation and

vice versa. This makes it difficult to talk about inputs and outputs, since it is not always clear if it is an input for the test case or for the implementation. Therefore we will use the terms *stimulus* for an output of the test case (i.e., an input of the implementation) and *response* for an input of the test case (i.e., an output of the implementation).

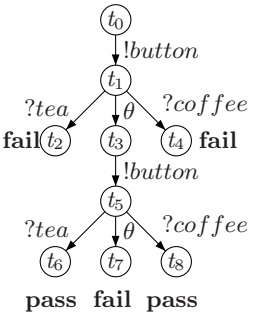


Fig. 7.10. Example test case

Example. In Figure 7.10 we show an example test case. With this test case we can test our coffee machine as specified, for example, in Figure 7.6. We see that the test case starts with the stimulus *button* in state t_0. In state t_1 we choose to observe a response. The specification prescribes that there should be no output. So if we observe coffee, or tea we add a **fail** label, like in t_2 and t_4. If we observe quiescence (with θ) we continue testing. Again we choose stimulus *button* and arrive in state t_5. Now the specification prescribes that we should observe coffee or tea as response. so, if we observe quiescence we add the **fail** verdict to state t_7. If we observe coffee or tea we stop testing and add **pass** as a verdict to states t_6 and t_8.

Test execution A test run of a test case on an implementation is modeled by synchronous parallel composition (denoted by $\|$) of the test case with the implementation under test. This means that inputs of the test case synchronize on outputs of the implementation and vice versa. In case of quiescence, the test case synchronizes on δ with its special θ action. The execution continues until the test case reaches one of its terminal nodes. Because of the special structure of a test case we are sure that the test case will always reach one of its terminal states. An implementation passes the test run if the test case ends in a pass state, if it is not we say that the implementation fails the test case. This means that we have found a possible error. Because an implementation can have non-deterministic behavior, different runs with the same test case can lead to different terminal states (and possibly different verdicts). Therefore, an implementation passes a test case if *all* possible test runs lead to the verdict pass.

Definition 7.32. Let $t \in \mathcal{TEST}(U, I)$ and $i \in \mathcal{IOTS}(I, U)$.

(1) A *test run* of a test case t with an implementation i is a trace of the synchronous parallel composition $t \| i$ leading to a terminal state of t:

$$\sigma \text{ is a test run of } t \text{ and } i \text{ iff } \exists i' : t \| i \stackrel{\sigma}{\Longrightarrow} \mathbf{pass} \| i' \text{ or } t \| i \stackrel{\sigma}{\Longrightarrow} \mathbf{fail} \| i'.$$

(2) Implementation i **passes** test case t if all their test runs lead to the pass-state of t:

$$i \textbf{ passes } t =_{\text{def}} \forall \sigma \in L^*_\theta, \forall i' : t \| i \stackrel{\sigma}{\not\Longrightarrow} \mathbf{fail} \| i'.$$

(3) An implementation i passes a test suite T if it passes all test cases in T:

$$i \textbf{ passes } T =_{\text{def}} \forall t \in T : i \textbf{ passes } t.$$

If i does not pass the test suite, it fails: i **fails** $T =_{\text{def}} \exists t \in T : i$ **pas⁄ses** t.

Test derivation All the parts of the conformance testing framework are now in place: a conformance relation between implementations and specifications and the execution of a test case on an implementation. We will finish the picture with test derivation (also called test generation). It is especially important that a test case is sound, i.e., if an implementation fails a test case it should be the case that there is really an error according to the specification. If possible we also want to generate a test suite that is exhaustive, i.e., if an implementation has an error then the test suite will detect it. In practice the latter is often impossible because of the (practically) infinite size of the test suite. Below we give the formal definitions of *completeness, soundness and exhaustiveness*. In this definition, we use **ioco** as the implementation relation. **ioco** can be replaced by any of the implementation relations of the previous section.

Definition 7.33. Let s be a specification and T a test suite then:

T is complete $=_{\text{def}} \forall i : i$ **ioco** $s \Leftrightarrow i$ **passes** T
T is sound $=_{\text{def}} \forall i : i$ **ioco** $s \Rightarrow i$ **passes** T
T is exhaustive $=_{\text{def}} \forall i : i$ **ioco** $s \Leftarrow i$ **passes** T

It turns out that a relative simple algorithm can produce a complete test suite for **ioco**. Test generation algorithms for the other implementation relations can be made in a similar way. For the completeness proof we refer to Tretmans [Tre96b]. Note that completeness often means an (practically) infinite test suite (one loop makes a complete test suite infinite). In the definition of the test derivation algorithm we have chosen to use a behavioral definition to make it easier to read (behavioral expression can be transformed to an LTS in a straightforward manner). This means that we do not explicitly create an LTS. To make it easier to understand the relation between the behavior and the LTS we added pictures to represent the way a test case is build up (so the pictures are test cases). Furthermore we give an example of a test case derivation after the definition. Note that we use the notation $\overline{\sigma}$ for a trace in which all δ actions have replaced by θ actions and all input actions have been changed to output actions (only the direction).

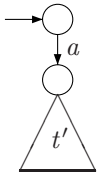
Definition 7.34. Let $s \in \mathcal{LTS}(I, U)$ be a specification with initial state q_0. Let S be a non-empty set of states, with initially $S = \{q_0\}$. A test case $t \in \mathcal{TEST}(U \cup \{\theta\}, I)$ is obtained from S by a finite number of recursive applications of one of the following three non-deterministic choices:

(1) 

$t :=$ **pass**

The test case with only the state **pass** is always a sound test case. This rule stops the recursion in the algorithm.

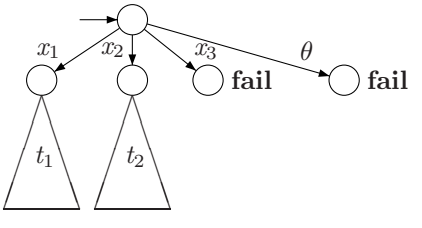
(2)



$t := \overline{a};\ t'$ where $a \in I, S$ **after** $a \neq \emptyset$ and t' is obtained by recursively applying the algorithm for $S' = S$ **after** a.

This step in the algorithm adds an input action a to the test case. After applying the input a, the test case behaves as t' which is obtained by applying the test derivation algorithm recursively to S'. t' is depicted as an abstract subtree (triangle) in the figure above.

(3)



$$
\begin{aligned}
t := \quad & \Sigma\,\{\overline{x};\ \textbf{fail} \mid x \in U, x \notin out(S)\} \\
& + \Sigma\,\{\theta;\ \textbf{fail} \mid \delta \notin out(S)\} \\
& + \Sigma\,\{\overline{x};\ t_x \mid x \in U, x \in out(S)\} \\
& + \Sigma\,\{\theta;\ t_\theta \mid \delta \in out(S)\}
\end{aligned}
$$

where t_x and t_θ are obtained by recursively applying the test derivation algorithm for $S' = S$ **after** x, S **after** δ, respectively. In this definition $+$ and Σ have the standard process algebraic meaning. $+$ stands for choice and Σ stands for the sum of all expressions in a set.

In this step we add expected outputs to the test case. If the output is incorrect according to the specification we add a transition with the output to a fail state, thus ending that part of the test case. The same holds for

the observation of quiescence where it is not allowed according to the specification. For outputs that are allowed, we continue the test derivation with t_x, t_θ respectively.

Example. We illustrate the test derivation algorithm with our coffee machine specification like the one shown on the left hand side in figure 7.6. We will show how to derive an LTS with the algorithm. We use the same test case as used to explain the test case definition, see Figure 7.10. When we start, the set S consists of only the start state q_1 of our specification. We choose one of the three rules of the test derivation algorithm. Randomly, we start with applying rule 2 of the test derivation algorithm and apply the input *button*. This is possible, since q_1 **after** $button = \{q_2\} (\neq \emptyset)$. The result is the transition $(t_0, !button, t_1)$ in our test case. The set S is updated to $S = \{q_2\}$. So we now have a test case with the stimulus *button*. Now we choose to observe responses from the implementation under test; we apply rule three. There are three possible responses: *tea*, $coffee$ and quiescence. We compute $out(q_1) = \{\delta\}$ of the specification. So, the only allowed output is quiescence. Therefore we add a transition with *tea* to a fail-state $(t_1, ?tea, t_2)$ and a transition with $coffee$ to a fail state $(t_1, ?coffee, t_4)$. For the allowed response δ we add the transition (t_1, θ, t_3). We update S with S **after** $\delta = \{q_2\}$. We again apply the stimulus *button* which results in the transition $(t_3, button, t_5)$ and $S = \{q_3\}$. For rule 3 there are now two options, either the response coffee or tea, since $out(q_2) = \{coffee, tea\}$. We can add the transitions $(t_5, ?tea, t_6)$, $(t_5, ?coffee, t_8)$ and (t_5, θ, t_7) where t_7 is a fail-state. Because of non-determinism in the specification we have two possible paths to continue with. For the "tea" path we update S with $\{q_5\}$ and for the "coffee" path we update S with $\{q_4\}$. We can in principle continue forever with choosing between step 2 and three of the test derivation algorithm until we reach a final state in the specification or until we want to stop. In our case the specification has reached a final state and we can apply rule 1 to stop the recursion. This transforms states t_6 and t_8 into pass-states.

7.5.1 Conformance Testing Based on Input/Output State Machine

In practice, system testing is performed with test suites. Each test case of a test suite is defined to verify that a specific property of the specifications is correctly implemented, or to detect a precise fault in the implementation. A test case can be seen as a finite sequence of interactions between a tester and the tested implementation. This process ends by the assignment of a verdict (usually pass, fail or inconclusive).

The method of conformance testing introduced by M. Phalippou on IOSM is different [Pha93]. Indeed, his idea consists in considering in a global way the set of all interactions and sequences of interactions between the tester and the tested implementation. According to this principle, a unique object, called canonical tester, is defined to represent in the one hand all the executions performed by a given test suite, and in the other hand all its execution.

To implement this approach, it is necessary to define concretely what is a canonical tester, as well as the way of assigning a test verdict with the couple (tester, implementation).

To ensure homogeneity with the specifications and the implementations, the canonical testers are modelled with IOSM. The canonical tester depends directly in the one hand on the implementation relation to be tested, and in the other hand on the trace machine of the specification. The trace machine of a specification S is a deterministic IOSM not comprising any internal action τ and having the same set of traces as the initial IOSM of S.

Definition 7.35 (Trace Machine)**.**
The trace machine of an IOSM $S = \langle S_s, L_s, T_s, s_{0s}\rangle$, noted $TM(S)$, is an IOSM $TM(S) = CC(\langle S_t, L_t, T_t, s_{0t}\rangle)$ defined by:

(1) S_t is the set of subsets of S_s: a state s_t of the trace machine is thus a set of states of the specification $s_t = \{s_{is}\}_{1\leq i\leq n}$
(2) $L_t = L_s$
(3) $s_{0t} = \{s \mid (s_{0s}, \varepsilon, s)\}$
(4) the transitions of the trace machine are exactly those obtained in the following way: for all $s \in S_t$ and $\mu \in \{!, ?\} \times L_s$, given $s' = \{s_j \mid (\exists\, s_{is} \in S_s)(s_{is}, \overrightarrow{\mu}, s_j)\}$, if $s' = \emptyset$ then $(s, \mu, s') \in T_t$.

The trace machine generation is similar to the determination of an not-deterministic automaton as introduced by J. Hopcroft and J. Ullman in [HU79]. Thus, from any IOSM, it is possible to calculate a trace machine, which exactly represents the traces of the initial IOSM.

Property 7.36. $Tr(S) = Tr(TM(S))$

The mechanism of verdict assignment is based on an parallel execution of the canonical tester with the implementation to be tested. The verdict is then pronounced according to the properties of the IOSM which represents this parallel composition. Indeed, the canonical tester has one particular state, called *fail*, which indicates that an error has been detected.

Definition 7.37 (Verdict of a canonical tester)**.**
The failure of a tester T applied to an implementation I is defined by: $Fail(T, I)$ iff $(\exists\, \sigma \in Tr(T))(\overrightarrow{\sigma} \in Tr(I) \wedge (T, \sigma, fail))$.

The verdict is $Succ(T, I)$ iff $\neg(Fail(T, I))$ holds.

The verdict assigned by the canonical tester is also defined as a global (or total) verdict.

We now present the test theory proposed by M. Phalippou using a concrete example. This example is based on the specification of the coffee machine example. The specification S introduced in figure 7.8 and implementations I_1, I_2, I_3 and I_4 introduced in figure 7.9 are used to illustrate the various steps to apply this testing theory.

First of all, we need to define the canonical tester making it possible to distinguish the IOSM which are implementations in conformity with the initial specification within the meaning of a specific implementation relation. The next definition describes how generate such a tester from the specification S of the coffee machine example and the implementation relation R_1 introduced by the definition 7.26.

Definition 7.38 (Canonical tester for R_1)**.**
Given a specification S and its trace machine $TM(S) = \langle S_s, L_s, T_s, s_{0s} \rangle$, we call canonical tester of S the IOSM noted $T = TCA(S) = \langle S_t, L_t, T_t, s_{0t} \rangle$ such as:

(1) $S_t = S_s \cup \{fail\}$
(2) $L_t = L_s$
(3) $s_{0t} = s_{0s}$
(4) the tester transitions are exactly those obtained by the following rules:
   (a) $(\forall\, \mu \in \{!, ?\} \times L_s)(\forall\, s, s' \in S_s)((s, \mu, s') \in T_s \Leftrightarrow (s, \overrightarrow{\mu}, s') \in T_t)$
   (b) $(\forall\, s \in S_s)(\forall\, a \in L_s)(\neg(\exists\, s')((s, !a, s') \in T_s) \Rightarrow (s, ?a, fail) \in T_t)$

This canonical tester of S is thus charged to check that nothing of opposite with what is envisaged can appear. For that, it provides an mirroring image of the traces of the specification. A mechanism to detect the errors is added: if the tester receives one interaction which should not happen in a given state, it reaches the state called *fail*. We can find this detection method in many approaches concerning testing from systems communicating by inputs and outputs, namely an inversion of the inputs and outputs to obtain tests from the specification [RP92]. The structure obtained by inversion of the trace machine determines the tester. This one is then supplemented by adding transitions used to detect errors.

The figure 7.11 introduces the canonical tester on the coffee machine example using the implementation relation R_1. For this example, the set L of all the events which can be received is $L = \{soup, tea, coffee\}$.

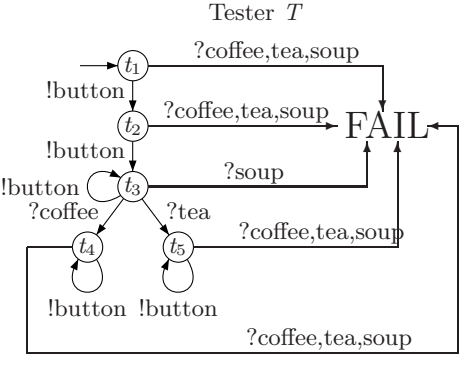


Fig. 7.11. Canonical tester using R_1

In order to assign a verdict, we now introduce the definition of the parallel composition of two IOSM. In fact, this composition makes it possible to model several communication interfaces between various IOSM or several points of interactions.

Definition 7.39 (Parallel composition of two IOSM)**.**
The parallel composition of two IOSM $I_1 = \langle S_1, L_1, T_1, s_{01}\rangle$ and $I_2 = \langle S_2, L_2, T_2, s_{02}\rangle$ is an IOSM $I = I_1 \,||\, I_2 = \langle S, L, T, s_0\rangle$ which is defined by:

(1) $S = S_1 \times S_2$
(2) $L = L_1 \cup L_2$
(3) $s_0 = (s_{01}, s_{02})$
(4) the transitions of the IOSM $I_1 \,||\, I_2$ are exactly obtained by means of the following rules:
   (a) $(s_1, \tau, s_1') \in T_1 \Rightarrow \forall\, s_2 \in S_2 \cdot ((s_1, s_2), \tau, (s_1', s_2)) \in T$
   (b) $(s_2, \tau, s_2') \in T_2 \Rightarrow \forall\, s_1 \in S_1 \cdot ((s_1, s_2), \tau, (s_1, s_2')) \in T$
   (c) if $a \in L_1 \cap L_2$ then
   $(s_1, ?a, s_1') \in T_1 \wedge (s_2, !a, s_2') \in T_2 \Rightarrow ((s_1, s_2), \tau, (s_1', s_2')) \in T$ and
   $(s_1, !a, s_1') \in T_1 \wedge (s_2, ?a, s_2') \in T_2 \Rightarrow ((s_1, s_2), \tau, (s_1', s_2')) \in T$
   (d) if $a \in L_1 - L_2$ then
   $(s_1, ?a, s_1') \in T_1 \Rightarrow \forall\, s_2 \in S_2 \cdot ((s_1, s_2), ?a, (s_1', s_2)) \in T$ and
   $(s_1, !a, s_1') \in T_1 \Rightarrow \forall\, s_2 \in S_2 \cdot ((s_1, s_2), !a, (s_1', s_2)) \in T$
   (e) if $a \in L_2 - L_1$ then
   $(s_2, ?a, s_2') \in T_2 \Rightarrow \forall\, s_1 \in S_1 \cdot ((s_1, s_2), ?a, (s_1, s_2')) \in T$ and
   $(s_2, !a, s_2') \in T_2 \Rightarrow \forall\, s_1 \in S_1 \cdot ((s_1, s_2), !a, (s_1, s_2')) \in T$

The IOSM calculated by parallel composition of the R_1 canonical tester and the possible implementations (figure 7.12) show well that the assignment of the verdict consists in checking if a state of the form $(fail, s_i)$ is in the IOSM $T \,||\, I_i$. So, the four generated IOSM confirm that, using the implementation relation R_1, only the implementation I_3 does not conform to the specification S.

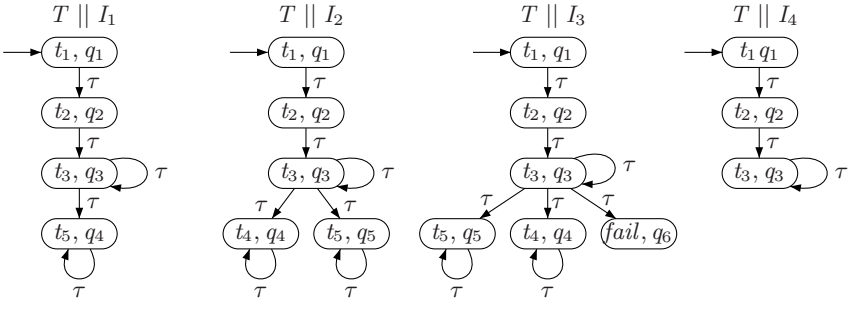


Fig. 7.12. Parallel composition tester/implementation

This global testing approach, already used by E. Brinksma in [Bri89] and introduced by M. Phalippou on IOSM, has the advantage of representing in a

homogeneous way the testing activity without having to pass by the stages of test suite generation and test case execution as shown in [Pha94a, Pha95].

However, it should be recognized that this approach presents disadvantages which are the counterpart of the advantages given above. It is for example difficult, without efficient test hypotheses, to study the concepts which are not defined in the specification, in particular how to connect the test cases to a test goal, how to add tests to increase the functional coverage of a test suite, or how to associate the test verdict with a specific diagnosis.

Finally, this method is used in the tool called TVEDA [Ris93, BPR93, CGPT96]. This tool is developed and used by the telecommunication industry (France Telecom) to automate the design of the test cases for protocol systems [Pha91]. In this area, the complexity of the systems, together with the high level of fiability which is expected from their global interworking, indeed justify to bring a great care to the test generation. In this way, TVEDA makes it possible to select a reasonable number of test cases by making some test hypotheses. Thus, the approach adopted in TVEDA slightly differs from the theory presented in this section: indeed, it consists, for a given implementation relation, to calculate an approximation of the tester since a rigorous definition is very often too complex to calculate (see Section 14.2.7 for more details).

7.6 Conclusion

In this chapter we filled in several pieces of the conformance testing framework for LTS-based testing with inputs and outputs. We started with the introduction of three models that capture the notion of inputs and outputs: the Input-Output Automaton, the Input-Output State Machine and the Input Output Transition System. An interesting characteristic of these models is that they are *input-enabled*. Next, we have shown several implementation relations over these models. The most important ones are: the fair testing preorder, the may and must preorder, the **ioco** implementation relation and the R_5 implementation relation. For the **ioco** theory and the theory of Phalippou we have shown how to derive test cases and how to execute them against an implementation. For the **ioco** theory there is a completeness proof for the test generation algorithm.

Finally, it should be stressed that these works are not simply regarded as significant theoretical results, but their practical applications directly contributed to the development of tools. Thus, an algorithm rising from the **ioco** theory is implemented in the tool TorX [dVT98], while the R_5 implementation relation is the base of the tool TVEDA [CGPT96].